The TL Compiler

Daniel Lungu*

Institute for computer Communications and Applications
École Polytechnique Fédérale de Lausanne

February 2000

Abstract

This paper describes a method of compiling *specification* properties into automata or test oracles which allow testing of any proposed *implementation*. It also shows how to interpret the automata's status during testing. The method addresses verification of reactive systems, including distributed object oriented applications.

TL (Temporal Logic) [4] has been chosen as specification language. The basic future temporal operators and logical operators are supported.

The automaton generated from a specification property (also called behavioural constraint) does not depend on the semantics of input events, such that the TL compiler can be used as a module in any verification system. The verification tool could be using a system description language to express proposed implementations [4] or run-time execution traces of the actual implementation [2].

* daniel.lungu@epfl.ch

1 Temporal Logic

Temporal logic is a language for specifying properties of reactive systems. Temporal logic extends the boolean predicate logic with a set of temporal operators that provide a succinct and abstract description of invariance, precedence or frequent recurrence of events in time.

An underlying assertion language for describing events in the system is assumed. A boolean formula in the assertion language (see §3) is referred as a *state formula*. For a state formula p and a state s that evaluates all the events in p, $s \Vdash p$ denotes that p holds on s (p is true on s).

1.1 Operators

A *temporal formula* (a *formula*) is constructed out of state formulas connected by means of boolean operators $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication) and the temporal operators presented in Table 1. Binary temporal operators get higher binding power than the binary boolean ones, in temporal formulas that are not fully parenthesised, as turns out from the grammar presented in Figure 1.

formula	spelling
$\Box p$	always p
$\Diamond p$	eventually p
$\bigcirc p$	next p
$p \mathcal{U} q$	p until q
$p \mathcal{W} q$	p waiting-for q

Table 1: Future temporal operators.

1.2 Semantics of Temporal Formulas

Temporal formulas are interpreted over a *model*, which is an infinite sequence of states $\sigma : s_0, s_1, \ldots, s_i, \ldots$ All the possible execution traces of the system are models. Given a model σ and a temporal formula p, $(\sigma, i) \models p$ denotes p holding at a position $i \geqslant 0$ in σ:

$$(\sigma, i) \models p \;\Leftrightarrow\; s_i \Vdash p$$

By using the future-tense operators in Table 1, one can easily describe specification properties without the aid of past-tense operators. The semantics of boolean and temporal operators are presented in Table 2, where p, q are eventually state formulas, at the bottom end of recursive substitution. In the general case, p, q of Table 2 may be temporal formulas being recursively substituted, according with the grammar shown in Figure 1.

Table 2: Operators semantics.

formula	semantics
$(\sigma, i) \models \neg p$	$\neg((\sigma, i) \models p)$
$(\sigma, i) \models p \wedge q$	$((\sigma, i) \models p) \wedge ((\sigma, i) \models q)$
$(\sigma, i) \models p \vee q$	$((\sigma, i) \models p) \vee ((\sigma, i) \models q)$
$(\sigma, i) \models p \rightarrow q$	$\neg((\sigma, i) \models p) \vee ((\sigma, i) \models q)$
$(\sigma, i) \models \Box p$	$(\sigma, j) \models p, \forall j \geqslant i$
$(\sigma, i) \models \Diamond p$	$\exists j \geqslant i, (\sigma, j) \models p$
$(\sigma, i) \models \bigcirc p$	$(\sigma, i+1) \models p$
$(\sigma, i) \models p \mathcal{U} q$	$(\exists k \geqslant i, (\sigma, k) \models q) \wedge ((\sigma, j) \models p, \forall i \leqslant j < k)$
$(\sigma, i) \models p \mathcal{W} q$	$((\sigma, i) \models p \mathcal{U} q) \vee ((\sigma, i) \models \Box p)$

2 TL Parser

The TL parser builds the syntactical tree for a formula. The grammar to be followed when writing specification properties is presented in Figure 1. Instead of providing the original symbols for operators (see §1.1), equivalent atoms from the ASCII set are proposed. For instance, `~` stands for ¬, or `[]` stands for □.

PropertyList ::= { *assignment* | *property* | *comment* }
assignment ::= *identifier* '`=`' *string*
property ::= '{' *expression* '}'
comment ::= '`//`' { *char* } *eol* | '`%`' { *char* } '`%`'
expression ::= *expression* '`->`' *disj* | *disj*
disj ::= *disj* '`|`' *term* | *term*
term ::= *term tlop conj* | *conj*
conj ::= *conj* '`&`' *factor* | *factor*
factor ::=
    '`(`' *expression* '`)`' |
    '`~`' *factor* | *utlop factor* |
    identifier | *string*
tlop ::=
    '`U`' | '`until`'
    '`W`' | '`waiting`' | '`unless`'
utlop ::=
    '`[]`' | '`always`' | '`henceforth`' |
    '`<>`' | '`eventually`' |
    '`o`' | '`next`'
identifier ::= *alpha* | '`_`' { *alnum* | '`_`' }
string ::= '`"`' { *char* } '`"`'
alpha ::= '`A`' .. '`Z`' | '`a`' .. '`z`'
alnum ::= *alpha* | '`0`' .. '`9`'

Figure 1: The TL grammar.

The precedence and associativity rules for TL operators are derived in Table 3. In the grammar's description, *char* stands for any character, *alpha* stands for letters and *alnum* stands for letters and digits. The specification language can be further enriched with other temporal operators by updating *tlop* and *utlop* nonterminals. The comments are ignored and may be used inside a property definition or assignment as well, which is not being shown by grammar.

operators	associativity
`( )`	left to right
`~ [] <> o`	right to left
`&`	left to right
`U W`	left to right
`|`	left to right
`->`	left to right

Table 3: Associativity and precedence of TL operators.

The events are verification system dependent and described by either a *string* or an *identifier*. The event syntax and semantic does not concern the TL compiler but the event parser (see §3) and verification system respectively.

An event can be described once by an *assignment* clause and referred from all subsequent *property* clauses by the mean of corresponding identifier. Nevertheless, the same *string* may be used several times for specifying properties, the event parser hopefully being able to figure out that the same event is addressed each time by looking up the event table.

When parsing the input stream, a *string* which has not been previously assigned to an *identifier* may be found. An unique identifier (e.g. `_ev1`, `_ev2`, etc.) must be automatically assigned to it. The following identifiers have been reserved as keywords and cannot designate events:

```
U       eventually  o
W       henceforth  unless
always  next        until
                    waiting
```

3 Event Parser

The event parser is in charge of analysing the content of a *string* encountered by the TL parser in the input stream. The valid event descriptors are stated by the next grammar, shown in Figure 2. The event grammar fits the required features of the MOTEL run time verification system, as described in [2]. MOTEL allows the testing of TL properties in a CORBA distributed application, by collecting at a central observer, notifications regarding a set of predefined types of basic events.

```
Event ::= expression relop expression | basic-event
expression ::= expression addop term | term
term ::=
      '(' expression ')' |
      '-' term | '+' term |
      '#' basic-event |
      integer
basic-event ::= type [ '(' args ')' ] | identifier
type ::=
      'o_outReq' | 'o_inReq' | 'o_outRep' | 'o_inRep' |
      's_newP' | 's_delP'
args ::= [ arg { ',' arg } ]
arg ::= identifier | '?' any-value '?' | '*'
identifier ::= alpha | '_' { alnum | '_' }
alpha ::= 'A' .. 'Z' | 'a' .. 'z'
alnum ::= alpha | '0' .. '9'
addop ::= '+' | '-'
relop ::= '<' | '<=' | '>' | '>=' | '=' | '<>'
```

Figure 2: The MOTEL event grammar.

In the grammar's description, `#` stands for number of occurrences of a basic event. A basic event is raised at run time by filters distributed across the system. There are six types of events supported by the current implementation of MOTEL, which are given by the *type* nonterminal. In Figure 2, `*` stands as a wild-card, `<>` stands for not equal with[1] and *any-value* stands for a character string corresponding to a CORBA marshaled value of an operation argument. This conversion is performed by the MOTEL filters before raising an event. The number and semantic of arguments for a specific event depend on the type of that particular event and on the IDL prototype of addressed operation, if any. The list of arguments might be omitted or trimmed, in which case all the missing arguments are set by default to `*` (match anything).

operators	associativity
`( )`	left to right
`# + -` (unary)	right to left
`+ -`	left to right
`< <= > >= = <>`	not defined

Table 4: Associativity and precedence of event operators.

The assertion language given in Figure 2 can be further extended or updated as the verification system's features evolve. The precedence and associativity rules for event grammar's operators are derived in Table 4.

[1] not to be confounded with TL's eventually

As an example, consider the `o_outReq` event type which occurs when an object sends a remote method invocation request. The associated state formula requires a variable list of arguments, according to the IDL operation's description:

$$\texttt{o\_outReq}(src, dst, op, arg_1, arg_2, \ldots)$$

The first three arguments should be specified by an *identifier* or `*`, designating the source, the target and the name of operation being invoked. The values of operation's parameters $arg_1, arg_2, \ldots$ may be subsequently fully or partially specified.

4 Generating Oracles

Temporal specifications describe constraints on the order of events occurring in executions of a concurrent application. The oracle generated from a temporal specification formula consists of a finite state automaton that accepts exactly those state sequences σ satisfying the specification. During testing, an oracle runs the automaton on state sequences induced by execution traces.

4.1 Preliminaries

A temporal formula f is checked at every state s_i within a sequence of states σ, by the mean of evaluating state formulas and performing the appropriate transitions in the corresponding automaton. State formulas are boolean expressions in an assertion language as the one given by the grammar of Figure 2. The definitions of state formulas determine events that must be monitored during testing, in order to produce execution traces. In the case of run-time test executions of concurrent systems, a state sequence σ induced by an execution trace is:

- **finite**, since test execution is finite
- **non-empty**, since test execution has an initial state

location formula	set of transitions
$f_1 \wedge f_2$	$\langle\{f_1, f_2\}\rangle$
$f_1 \vee f_2$	$\langle\{f_1\}\{f_2\}\rangle$
$\Box f$	$\langle\{f, \bigcirc\Box f\}\rangle$
$\Diamond f$	$\langle\{f\}\{\odot\Diamond f\}\rangle$
$f_1 \mathcal{U} f_2$	$\langle\{f_2\}\{f_1, \odot(f_1 \mathcal{U} f_2)\}\rangle$
$f_1 \mathcal{W} f_2$	$\langle\{f_2\}\{f_1, \bigcirc(f_1 \mathcal{W} f_2)\}\rangle$

Table 5: Reduction rules for TL operators.

The semantics of TL operators is captured by the reduction rules given in Table 5, being used when compiling temporal formulas. A reduction rule associates a formula with a sequence of set of formulas or list of *options* implying it.

Intuitively, the options associated with a formula f, represent different ways of satisfying it. In terms of generated automaton, they model a set of potential transitions outgoing from a location satisfying f.

For example, the reduction rule for $\Box$ defines a single option $\{f, \bigcirc\Box f\}$ stating that $(\sigma, i) \models \Box f$ if $(\sigma, i) \models f$ and either $(\sigma, i+1) \models \Box f$ or (σ, i) is the final state of σ. The reduction rule for $\Diamond$ defines two options: the former, $\{f\}$ stating that $(\sigma, i) \models f \Rightarrow (\sigma, i) \models \Diamond f$; the latter, $\{\odot\Diamond f\}$ stating that $(\sigma, i+1) \models \Diamond f \Rightarrow (\sigma, i) \models \Diamond f$, when (σ, i) is not the final state of σ, as expressed by the newly introduced $\odot$ (strong next) operator. The strong next operator is only used in compiler internally generated formulas.

The automaton[2] generated from a temporal formula is a bipartite flow graph comprising *locations*, shown as circles, and *transitions*, depicted as rectangles (see Figures 4, 5 or 6). A pair of arcs $(L, T), (T, L')$, with T a flow graph transition and L, L' flow graph locations stands for the transition (L, T, L') in the conventional finite state automaton representation. There is always an initial location, called L_0 and several final locations denoted by a double circle.

The input alphabet of an automaton is the set of all possible boolean vectors of state formula valuations. A transition T is annotated with the state formula which enables it. If T is anyhow enabled, it is annotated with *true*. The automaton built for a formula f accepts precisely those state sequences σ that satisfy f (associated with L_0). Thus, an oracle runs the automaton during testing to verify that $s_0 \models\!\!\!\models f$. When there are no enabled transitions outgoing from the current set of locations, the oracle rejects σ as violating f.

$\neg(\neg a)$	$=$	a	$\neg(\Box a)$	$=$	$\Diamond\neg a$
$\neg(a \wedge b)$	$=$	$\neg a \vee \neg b$	$\neg(\Diamond a)$	$=$	$\Box\neg a$
$\neg(a \vee b)$	$=$	$\neg a \wedge \neg b$	$\neg(\bigcirc a)$	$=$	$\odot\neg a$
$a \rightarrow b$	$=$	$\neg a \vee b$	$\neg(a\,\mathcal{U}\,b)$	$=$	$\neg b\,\mathcal{W}(\neg a \wedge \neg b)$
$\neg(a \rightarrow b)$	$=$	$a \wedge \neg b$	$\neg(a\,\mathcal{W}\,b)$	$=$	$\neg b\,\mathcal{U}(\neg a \wedge \neg b)$

Table 6: NNF rules for TL operators.

Definition 1 *A temporal formula f is in negation normal form (NNF) if negations apply only to state formulas and the implications have been replaced by disjunctions.*

For example, by applying the NNF rules given in Table 6, the normal form of $f = \neg(\Box(p \rightarrow q\,\mathcal{U}\,r))$, where p, q, r are state formulas, is derived as follows:

$$f = \Diamond\neg(p \rightarrow q\,\mathcal{U}\,r) = \Diamond\neg(\neg p \vee q\,\mathcal{U}\,r) = \Diamond(p \wedge \neg(q\,\mathcal{U}\,r)) = \Diamond(p \wedge (\neg r\,\mathcal{W}(\neg q \wedge \neg r)))$$

Since the temporal formulas are normalised into NNF, Table 5 includes neither reduction rules for negation forms of TL operators e.g. $\neg\Box f$, $\neg(f_1\,\mathcal{U}\,f_2)$, nor a reduction rule for $f_1 \rightarrow f_2$.

[2] nondeterministic in general

Definition 2 *An atomic formula has the form a or $\neg a$ for some state formula a. A next formula has the form $\bigcirc f$ or $\odot f$ for some temporal formula f. A formula is basic if it is an atomic formula or a next formula.*

Intuitively, atomic formulas are verified at the current state s_i of σ, and next formulas are verified at the next state s_{i+1}, if any. Since $\neg f$ and $\bigcirc f$ are basic formulas, there is no need of a corresponding reduction rule for $\neg$ or $\bigcirc$ in Table 5.

Sets of formulas, such as the only option of reduction rule for $\Box$, are interpreted as conjunctions, whereas a sequence of sets of formulas, such as the options of reduction rule for $\Diamond$, is interpreted as disjunction. For instance, the options of reduction rule for $\mathcal{U}$ denote the formula $f_2 \vee (f_1 \wedge \odot(f_1 \mathcal{U} f_2))$.

4.2 The Tableau Algorithm

Given a temporal formula f, the tableau algorithm constructs a nondeterministic automaton, i.e. a flow graph that accepts precisely those sequences of input vectors satisfying f. Each location is associated with a formula L_k, which must be verified by the remaining state sequence: $(\sigma, i) \models L_k$. For the initial location, $(\sigma, 0) \models L_0$. Subsequently, the location and transition names identify the associated formulas.

The first step on which the algorithm proceeds is the normalisation of f, in order to minimise the number of reduction rules to those presented in Table 5. The resulting formula L_0 is associated with the initial location. Derivation of NNF for temporal formulas appears in Appendix A. It recursively applies the rules given in Table 6 until atomic formulas are reached.

When L_0 is a *safety* formula i.e. has the form $\Box p$, it may be substituted with the NNF of $\neg L_0$ instead, such that the generated automaton will accept the sequences of input vectors violating f. This design option has been considered while writing the compiler.

The next step elaborates L_0 into a sequence of basic formula sets by applying the reduction rules presented in Table 5. The tableau algorithm is performed by function $elaborate(F, N, f_0)$ (see Figure 3), initially invoked as $elaborate(\emptyset, \emptyset, L_0)$ (see the implementation in Appendix B). The first two arguments are formula sets, such that $F \cup N$ is the set of formulas which must be verified by the location being elaborated, and N contains all and only the next formulas of $F \cup N$. The third argument, f_0 is a startup formula which, is first included in F (see line 2).

When $F \cup N$ consists solely of basic formulas (lines 3,4), a new transition T outgoing from the location being elaborated should be instantiated and installed into the flow graph (lines 9,12). The formula associated with T is the conjunction of formulas in F (line 5). The flow graph is described by the set of locations *Locs* and the set of transitions *Trans*, being recursively built in $elaborate()$. The new location L as destination of T is instantiated as well (line 4). The formula associated with L is the conjunction of next formulas in N (line 6) after removing the next operator, $\bigcirc$ or $\odot$. When N has no strong next formulas, L can be declared final (lines 7,8), otherwise L can not be final as long as semantic of $\odot$

precludes that. Location L is ready to be installed as destination of T (line 10) and included in the set *Locs* (line 14). Whenever a new location is included in the flow graph, it follows the same elaboration process by the recursive call in line 15.

```
Location :: elaborate(F, N, f_0 = true)
 1  if f_0 <> true then
 2      F := F ∪ {f_0}
 3  if ⋆ F has only atomic formulas then
 4      ⋆ let L be a new Location and T a new Transition
 5      T.formula := ⋀_{f∈F} f
 6      L.formula := ⋀_{f∈N} f.left
 7      if f.type = ◯, ∀f ∈ N then
 8          L.final = true
 9      next := next ∪ {T}
10      T.next := L
11      if T ∉ Trans then
12          Trans := Trans ∪ {T}
13      if L ∉ Locs then
14          Locs := Locs ∪ {L}
15          L.elaborate(∅, ∅, L.formula)
16  else
17      ⋆ let f ∈ F be a non-atomic formula
18      F := F \ f
19      switch f.type
20      ∧ :     F := F ∪ {f.left, f.right}
21              elaborate(F, N)
22      ∨ :     elaborate(F, N, f.left)
23              elaborate(F, N, f.right)
24      □ :     N := N ∪ {◯f}
25              elaborate(F, N, f.left)
26      ◇ :     elaborate(F, N, f.left)
27              N := N ∪ {⊙f}
28              elaborate(F, N)
29      ◯ ⊙ :   N := N ∪ f
30              elaborate(F, N)
31      𝒰 𝒲 :   elaborate(F, N, f.right)
32              if f.type = 𝒰 then
33                  N := N ∪ {⊙f}
34              else
35                  N := N ∪ {◯f}
36              elaborate(F, N, f.left)
```

Figure 3: The recursive tableau algorithm.

When $F \cup N$ has to be further reduce, i.e. there is a nonbasic formula $f \in F \cup N$ (line 17), the reduction rules given in Table 5 are applied in a straightforward manner (lines 20-36).

In order to simplify the presentation of the tableau algorithm, several assumptions have been made in Figure 3. In 5th line, if $F = \emptyset$, formula associated with T is *true*. A similar observation can be made for line 6, regarding L. In 9th line, the new transition T is added to the list `next` of outgoing transitions from the current location, only if it is not already a member of `next`. When adding a new formula to F or to N, a specific decreasing order is considered amongst the elements of a formula sets, such that for any formulas, f_1, f_2:

$$\begin{aligned} \mathit{true} &< f_1 \\ \mathcal{P}(f_1) = \mathcal{P}(f_2) &\Leftrightarrow f_1 = f_2 \\ \mathcal{P}(f_1) \subseteq \mathcal{P}(f_2) &\Rightarrow f_1 < f_2 \\ \mathcal{P}(f_1) < \mathcal{P}(f_2) &\Rightarrow f_1 < f_2 \end{aligned}$$

$\mathcal{P}(f)$ is the polish postfix translation of formula f i.e. the string obtained by traversing in postorder the syntactical tree describing f. The semantics of string comparison have been used. Two identical formulas $f_1 = f_2$ are not allowed to become members of the same set F (lines 2,20) or N (lines 24,27, ...). When picking a non-atomic formula from F in line 17, f is the first element of F, according to the previously defined order. Thus, when considering two non-basic formulas $f_1, f_2 \in F$, the one which a reduction rule is applied on is $f = max(f_1, f_2)$.

The algorithm elaborates all the locations *Locs* that can be reached from the initial one L_0, and generates the appropriate transitions *Trans*. Because the number of formulas that can be developed in this fashion is finite, only a finite number of locations can be produced, and the elaboration process eventually terminates.

4.3 Samples

Safety

The temporal formula $f = \Box p$ requires that state formula p is an invariant over state sequence σ. By applying the tableau algorithm on $\neg f$, the automaton shown in Figure 4 is produced. First, the NNF of $\neg f$ is derived, $L_0 = \neg\Box p = \Diamond\neg p$. Then L_0 is elaborated in a recursive manner:

$$L_0 = \langle\{\Diamond\neg p\}\rangle\} = \langle\{\neg p\}\{\odot\Diamond\neg p\}\rangle$$

Two basic formula sets have been obtained, corresponding to transitions T_1, T_3 outgoing from L_0. T_3 leads back to L_0 whereas T_1 reaches a final location L_1, associated with a void sequence of formula sets:

$$L_1\# = \langle\rangle$$

There is a transition T_2 looping on L_1 and annotated with *true*.

Next

The temporal formula $L_0 = \bigcirc p$ states that $(\sigma, 1) \models p$. L_0 is already in NNF form, therefore the elaboration process is initiated: $L_0 = \langle\{\bigcirc p\}\rangle$. This basic formula

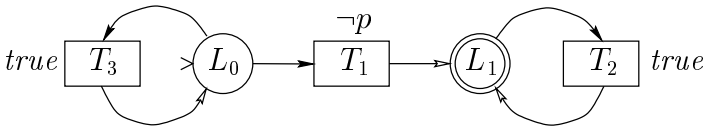

Figure 4: Generated automaton for $\neg(\Box p) = \Diamond\neg p$.

defines the *true* annotated transition T_1 in the automaton of Figure 5. It leads to a final location L_1 which is further elaborated, $L_1\# = \langle\{p\}\rangle$, yielding transition T_2 labelled with p. Transition T_2 leads to a final location L_2, associated with an empty sequence of formula sets, $L_2\# = \langle\rangle$. There is a transition T_3 looping on L_2 and labelled with *true*.

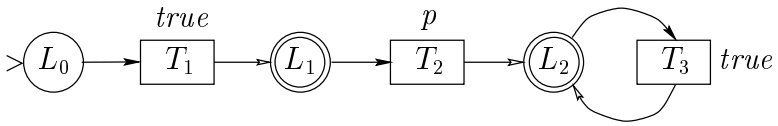

Figure 5: Generated automaton for $\bigcirc p$.

Precedence

The temporal formula $f = \Box(p \rightarrow q\,\mathcal{U}\,r)$ requires that any p-state initiates a q-interval in σ, terminated by an r-state. By applying the tableau algorithm on $\neg f$, the automaton shown in Figure 6 is produced. First, the NNF of $\neg f$ is derived, $L_0 = \neg f = \Diamond(p \wedge (\neg r\,\mathcal{W}(\neg q \wedge \neg r)))$, as already shown. Then L_0 is elaborated in a recursive manner:

$$
\begin{aligned}
L_0 &= \langle\{\Diamond(p \wedge (\neg r\,\mathcal{W}(\neg q \wedge \neg r)))\}\rangle \\
&= \langle\{p \wedge (\neg r\,\mathcal{W}(\neg q \wedge \neg r))\}, \{\odot\Diamond(p \wedge (\neg r\,\mathcal{W}(\neg q \wedge \neg r)))\}\rangle \\
&= \langle\{p, \neg r\,\mathcal{W}(\neg q \wedge \neg r)\}, T_6\rangle \\
&= \langle\{p, \neg q \wedge \neg r\}, \{p, \neg r, \bigcirc(\neg r\,\mathcal{W}(\neg q \wedge \neg r))\}, T_6\rangle \\
&= \langle\{p, \neg q, \neg r\}, T_3, T_6\rangle = \langle T_1, T_3, T_6\rangle \\
L_1\# &= \langle\rangle = \langle T_2\rangle \\
L_2\# &= \langle\{\neg r\,\mathcal{W}(\neg q \wedge \neg r)\}\rangle \\
&= \langle\{\neg q \wedge \neg r\}, \{\neg r, \bigcirc(\neg r\,\mathcal{W}(\neg q \wedge \neg r))\}\rangle \\
&= \langle\{\neg q, \neg r\}, T_5\rangle = \langle T_4, T_5\rangle
\end{aligned}
$$

Transition T_1 leads to the final state L_1, since there are no strong next formulas in $\{p, \neg q, \neg r\}$. The formula associated with L_1 is *true* which elaborates into a *true* annotated transition T_2. A location such L_1 is never leaved as soon as it is reached, when running the automaton. The automaton might receive an external reset and return to initial location. A final location with an infinite loop transition is called *error state* in the case of safety properties, and *accepting state* for non-safety properties.

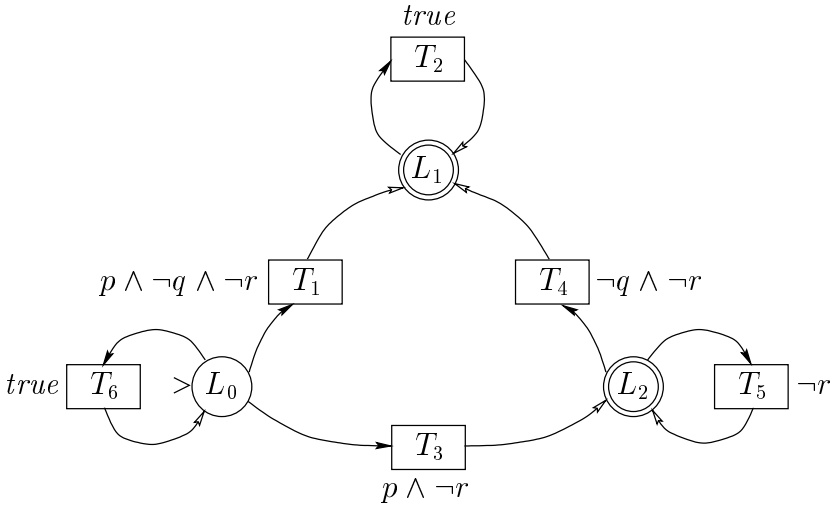

Figure 6: Generated automaton for $\neg(\Box(p \rightarrow q\,\mathcal{U}\,r)) = \Diamond(p \wedge (\neg r\,\mathcal{W}(\neg q \wedge \neg r)))$.

4.4 Internal Structure

A class hierarchy has been deployed in order to facilitate the interaction between the two modules dealing with TL grammar and event grammar respectively. Figure 7 presents the UML class diagram of the TL compiler.

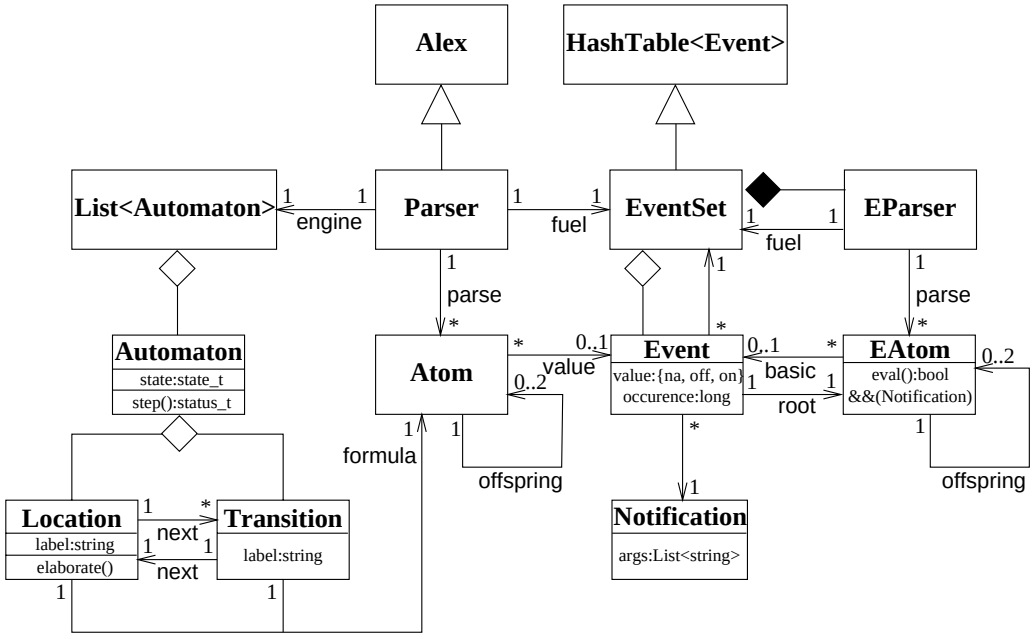

Figure 7: Class diagram.

Class `Notification` links the TL and observation modules in a verification tool. A *true* label has been represented as a nil reference i.e. `(Atom *)NULL`.

5 Testing

Since the generated flow graph is nondeterministic, the current state of the automaton is a set of locations, $S \subseteq Locs$. When a new event is detected, the automaton should perform the step to the next state S_{next}, by considering all $L \in S$ and following the transitions T which evaluate to *true*, as shown in Figure 5.

```
Automaton :: step()
    if safety then
        if squo = error then
            return squo
    else
        if squo = accept then
            return squo
    squo := good
    for ⋆ each L ∈ S do
        for ⋆ each T ∈ L.next do
            if T.formula.eval() = true then
                if T.next ∉ S_next then
                    S_next := S_next ∪ {T.next}
                if T.next.final = true then
                    if squo = good then
                        squo := final
                    if T.next.loop = true then
                        if safety then
                            squo := error
                        else
                            squo := accept
    if S_next = ∅ then
        if safety then
            squo := accept
        else
            squo := error
    return squo
```

Figure 8: Performing a step $S \rightarrow S_{next}$ in a nondeterministic flow graph.

Whenever the automaton reaches an accept or error state, it is locked and does not perform any further step unless an external reset is received: $S = \{L_0\}$. The user may choose to delete the automaton, since the corresponding property has been fullfiled or violated respectively.

Acknowledgement

I would like to address thanks to my colleagues, *Xavier Logean* and *Falk Dietrich* who have been patiently supporting this work.

A Normalise snapshot

The `normalise()` static method of class `Automaton` is given below.

```
void Automaton::negate(Atom *&p)
{
    if (p == NULL)
        return;
    switch (p->type) {
    case Atom::is_event :
        p = new Atom(Atom::op_not, p);
    case Atom::is_op :
        switch (p->value.op) {
        case Atom::op_and :
            p->value.op = Atom::op_or;
            negate(p->left);
            negate(p->right);
        ...
        }
    case Atom::is_uop :
        switch (p->value.uop) {
        case Atom::op_always :
            p->value.uop = Atom::op_eventually;
            negate(p->left);
        ...
        }
    }
}

void Automaton::normalise(Atom *&p)
{
    if (p == NULL)
        return;
    while (p->is(Atom::op_not)) {
        if (p->left->is(Atom::is_event))
            return;
        delete p;
        negate(p = p->left);
    }
    if (p->is(Atom::op_implies)) {
        negate(p->left);
        p->value.op = Atom::op_or;
    }
    normalise(p->left);
    normalise(p->right);
}
```

B Elaborate snapshot

The `elaborate()` method of class `Location` is given below.

```
void Location::elaborate(List <Atom *> fl, List <Atom *> nl, Atom *fins)
{
    Atom *f, *n;

    if (fins != NULL)
        insertf(fl, fins);
    if (!get_nonbasic(fl, f)) {
        insertt(fl, nl);
        return;
    }
    fl.del();
    switch (f->type) {
    case Atom::is_op :
        switch (f->value.op) {
        case Atom::op_and :
            insertf(fl, f->left);
            insertf(fl, f->right);
            elaborate(fl, nl, NULL);
        case Atom::op_or :
            elaborate(fl, nl, f->left);
            elaborate(fl, nl, f->right);
        ...
        }
    case Atom::is_uop :
        switch (f->value.uop) {
        case Atom::op_always :
            n = insertn(nl, Cookie::op_next, f);
            elaborate(fl, nl, f->left);
            delete n;
        case Atom::op_eventually :
            elaborate(fl, nl, f->left);
            n = insertn(nl, Cookie::op_snext, f);
            elaborate(fl, nl, NULL);
            delete n;
        case Atom::op_next : case Atom::op_snext :
            nl.put(f);
            elaborate(fl, nl, NULL);
        }
    }
}
```

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Falk Dietrich, Xavier Logean, and Jean-Pierre Hubaux. Testing temporal logic properties in distributed systems. In *IFIP International Workshop on Testing of Communicating Systems (IWTCS)*, pages 247–262, Tomsk, Russia, August 1998.

[3] L.K. Dillon and Y.S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, October 1996.

[4] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.